STUDENT PAPER: Revising and Expanding a Blue Waters Curriculum Module as a Parallel Computing Learning Experience

Ruth Catlett[1]
University of Mary Washington
1301 College Avenue
Fredericksburg, VA 22401
rcatlett@umw.edu

David Toth[2]
Centre College
600 West Walnut Street
Danville, KY 40422
David.toth@centre.edu

ABSTRACT

The party problem is a mathematical problem in the discipline of Ramsey Theory. Because of the problem's embarrassingly parallel nature, its extreme computational requirements, and its relative ease of understanding implementation with a naïve algorithm, it is well suited to serve as an example problem for teaching parallel computing. Years ago, a curriculum module for Blue Waters was developed using this problem. However, delays in the delivery of Blue Waters resulted in the module being released before Blue Waters was accessible. Therefore, performance data and compilation instructions for Blue Waters were not available. We have revised the module to provide source code for new versions of the programs to demonstrate more parallel computing libraries. We have also added performance data and compilation instructions for the code in the old version of the module and for the new implementations, which take advantage of the capabilities of the Blue Waters supercomputer now that it is available.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – parallel programming.

General Terms

Experimentation.

Keywords

Parallel computing education, Ramsey theory.

1. INTRODUCTION

The party problem is a problem in Ramsey Theory, an area of mathematics that focuses on "the mathematical study of combinatorial objects in which a certain degree of order must occur as the scale of the objects becomes large" [1]. The general form of the party problem, R(m, n) seeks to determine the number of people that must attend a party such that there is guaranteed to be a group of m people who all know each other, a group of n people who are all complete strangers, or both [2]. The R(5, 5) instance of the party problem is still unsolved, requiring immense computational power to solve [3]. We refer the reader to [4] for more information on the party problem and an algorithm used by some programs that attempt to solve it. In 2012, a curriculum module for teaching parallel computing using the party problem as an example was released [4]. While this module was designed for the Blue Waters supercomputer, Blue Waters was delayed and the module was released before the hardware, so it could not include compilation instructions and performance data for Blue Waters. While the module could still be used for teaching parallel computing, we have updated it significantly with the information specific to the Blue Waters supercomputer, including instructions to compile the all the code and performance data from Blue Waters. We have also added two additional versions of the programs, one using MPI and the other one which is an MPI/CUDA hybrid.

2. RELATED WORK

Toth and Bryant developed code to test 335,544,320,000 graphs for the party problem, producing sequential, OpenMP, and CUDA versions of the code [4]. While the code would not solve the party problem for the R(5, 5) instance due to the need to test more graphs than could be tested by a dedicated supercomputer in a lifetime, its three implementations that all use the same algorithm and the embarrassingly parallel nature of the problem made it a nice way to introduce students to parallel computing. Thus, the code formed the foundation for the curriculum module [5]. However, the lack of performance data from Blue Waters, a system that could be used by multiple people, and the lack of instructions to compile and run the code on Blue Waters made the module less useful than it would be with those features. We note that there are a number of other such modules available at http://www.shodor.org/petascale/materials/modules/ from a wide range of disciplines, but few have implementations in all of MPI, OpenMP, and CUDA [6].

3. MODULE UPDATES

For this update we looked at the existing module which included a sequential version, a two-file Compute Unified Device Architecture (CUDA) version, and an Open Multi-Processing (OpenMP) version. In addition, there were some instructions and many comments in the code. The institute held in Illinois at the beginning of this project taught how to use the CUDA, OpenMP, and Message Passing Interface (MPI) libraries as well as how to make hybrids with the libraries. The first steps of the code writing portion of the project were to write an MPI version and an MPI and CUDA hybrid. In writing the MPI and CUDA hybrid we found it was easier to use if the CUDA was in one file so we edited the CUDA version to be one file and added command line

[1] Undergraduate Student

[2] Corresponding Author

arguments so the user can specify the number of blocks and threads in the kernel call. We also made sure that every version was testing the same number of graphs and was creating the same output, printing out either a graph that did not have a K5 or, a statement saying that no such graph was found.

With all of these added versions we also had to add compilation and execution instructions for each version. We also tested different numbers of nodes and cores so that we could provide users accurate performance information. This also showed us how efficient each version was in with different numbers of nodes.

The different versions allowed us to highlight the different ways we can use the parallel hardware of Blue Waters. The MPI library allowed us to use multiple nodes on Blue Waters. MPI allowed us to start processes on multiple nodes and enabled them to coordinate their graph testing to divide the graphs to be tested and reduce the wall clock time to run the program. The CUDA library allowed us to divide the graphs among the GPU cores on a single node, and the MPI and CUDA hybrid allowed us to use multiple GPUs, enabling us to test the graphs in a very short amount of time.

4. METHODS

We conducted performance tests on Blue Waters for each of the programs. For the sequential, OpenMP, and MPI versions of the program, ten runs were done for each version and the average of the ten trials was taken. The performance of the CUDA version is dependent upon the number of threads and blocks that the program uses and there is no particular set of values that work for every program. Some people have stated that there should be at least 64 threads per block and that number should be a multiple of 64 [7]. Threads per block between 128 and 256 gave others the best performance for their applications [8]. Therefore, for the CUDA version of the program, we tested the program with different numbers of threads and blocks to determine the best values for those parameters. For the number of threads per block, we tested 4096, 8192, and 16384. For the number of blocks, we tested 64, 128, 256, 512, and 1024. Once we determined the best values for blocks and threads per block, we did the performance testing for the MPI and CUDA hybrid program using those values.

5. RESULTS

The times each of the ten runs took for the sequential version, the OpenMP version, and the MPI version are shown in Table 1 and Table 2. The times from the tests of the CUDA version that we used to determine which numbers of blocks and threads per block are shown in Tables 3-5. For the CUDA version, we found that 64 threads per block resulted in a runtime of over 1.5 times the runtimes using 128, 256, 512, and 1024 threads per block. Although the runtimes using the other numbers of threads per block and all of the numbers of blocks that we tried were close, 128 threads per block and 16,384 blocks produced the fastest average times. The results of the MPI/CUDA hybrid version of the program are shown in Table 6.

While the runtime of the OpenMP version of the program decreased as the number of CPU cores it used was increased as shown in Figure 1, the speedup achieved shown in Table 7 was not linear with the number of cores. This could be because different graphs take different times to examine. If the graphs that take a longer time are concentrated in a single or a couple sections of the graphs, then that could result in the speedup being less than linear. The speedups for the MPI program and the MIP and CUDA hybrid programs are shown in Table 8 and Table 9. Thos speedups show similar results to the OpenMP speedups, but are not quite as good. We expect that this is because in addition to the distribution of the graphs that take longer to examine, these programs also need to send information between nodes, which should result in a performance loss.

Table 1 - Runtime for Sequential and OpenMP Versions

Trial	Sequential	OpenMP Using 1 Core	OpenMP Using 2 Cores	OpenMP Using 4 Cores	OpenMP Using 8 Cores	OpenMP Using 16 Cores	OpenMP Using 32 Cores
1	17839	17794	12977	6425	3616	1801	917
2	17838	17720	12811	6416	3601	1815	912
3	17740	17684	12816	6438	3599	1808	906
4	17792	17666	12829	6426	3600	1804	908
5	17899	17631	12819	6410	3581	1800	910
6	17911	17679	12825	6487	3595	1802	914
7	17928	17813	12918	6417	3565	1806	912
8	17835	17641	12831	6526	3556	1811	905
9	18053	17743	12819	6413	3591	1810	911
10	17882	17778	12833	6415	3597	1819	907
Average	17871.7	17714.9	12847.8	6437.3	3590.1	1807.6	910.2

Table 2 - Runtimes for MPI Version

Trial	MPI Using 1 Node	MPI Using 2 Nodes	MPI Using 4 Nodes	MPI Using 8 Nodes	MPI Using 16 Nodes	MPI Using 32 Nodes
1	927	469	238	127	66	37
2	921	478	239	124	67	37
3	925	472	240	124	67	37
4	914	473	238	123	67	36
5	917	476	239	124	65	39
6	926	471	238	124	65	37
7	927	471	238	124	66	38
8	918	479	240	124	66	38
9	918	475	239	124	66	36
10	912	469	239	123	67	37
Average	920.5	473.3	238.8	124.1	66.2	37.2

Table 3 - Runtimes for CUDA Version with 4096 Blocks

Trial	64 Threads Per Block	128 Threads Per Block	256 Threads Per Block	512 Threads Per Block	1024 Threads Per Block
1	142	86	84	86	87
2	142	85	84	86	87
3	142	85	84	85	87
4	143	85	85	85	87
5	143	85	84	86	87
6	143	85	85	85	86
7	143	85	85	85	86
8	143	85	85	85	86
9	143	85	85	86	87
10	142	85	84	86	87
Average	142.6	85.1	84.5	85.5	86.7

Table 4 - Runtimes for CUDA Version with 8192 Blocks

Trial	64 Threads Per Block	128 Threads Per Block	256 Threads Per Block	512 Threads Per Block	1024 Threads Per Block
1	140	85	84	86	88
2	140	84	84	86	88
3	140	84	85	86	87
4	140	84	84	86	87
5	140	85	84	86	87
6	140	84	85	86	88
7	141	84	85	86	88
8	140	84	85	86	87
9	140	84	84	86	88
10	140	85	84	86	88
Average	140.1	84.3	84.4	86	87.6

Table 5 - Runtimes for CUDA Version with 16384 Blocks

Trial	64 Threads Per Block	128 Threads Per Block	256 Threads Per Block	512 Threads Per Block	1024 Threads Per Block
1	140	84	85	86	90
2	139	85	86	86	90
3	139	84	85	87	89
4	140	84	85	86	89
5	140	84	85	86	89
6	140	85	85	87	90
7	139	84	85	87	89
8	140	84	85	87	90
9	139	84	85	87	89
10	140	84	85	87	89
Average	139.6	84.2	85.1	86.6	89.4

Table 6 - Runtimes for MPI-CUDA Hybrid Version

Trial	2 Compute Nodes	4 Compute Nodes	8 Compute Nodes	16 Compute Nodes	32 Compute Nodes	64 Compute Nodes
1	43	22	12	7	4	3
2	43	22	12	7	3	3
3	43	22	12	6	4	3
4	43	22	12	7	4	2
5	43	22	11	6	4	2
6	44	22	12	6	4	3
7	43	22	12	7	4	3
8	43	22	12	6	5	3
9	43	22	11	7	4	3
10	43	22	11	7	4	2
Average	43.1	22	11.7	6.6	4	2.7

Table 7 - OpenMP Speedups and Efficiencies vs. Sequential Program

Cores	Speedup	Maximum Possible Speedup
1	1.0	1
2	1.4	2
4	2.8	4
8	5.0	8
16	9.9	16
32	19.6	32

Table 8 - MPI Speedups and Efficiencies vs. Sequential Program

Nodes	Speedup	Maximum Possible Speedup
1	19.4	32
2	37.8	64
4	74.8	128
8	144.0	256
16	270.0	512
32	480.4	1024

Table 9 - MPI/CUDA Hybrid Speedups vs. 1 Node CUDA Program

Nodes	Speedup	Maximum Possible Speedup
2	1.95	2
4	3.83	4
8	7.20	8
16	12.73	16
32	20.87	32
64	30.90	64

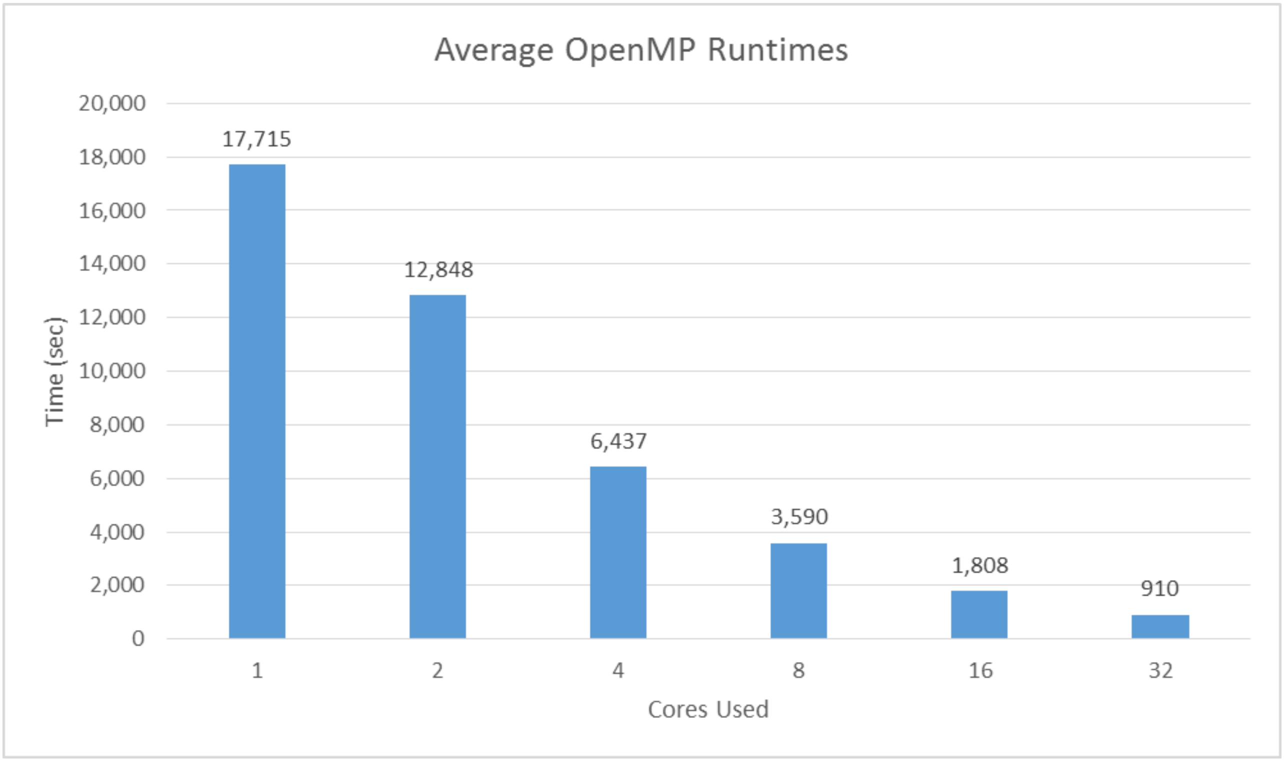

Figure 1 - Average Runtimes vs. Cores Used with OpenMP

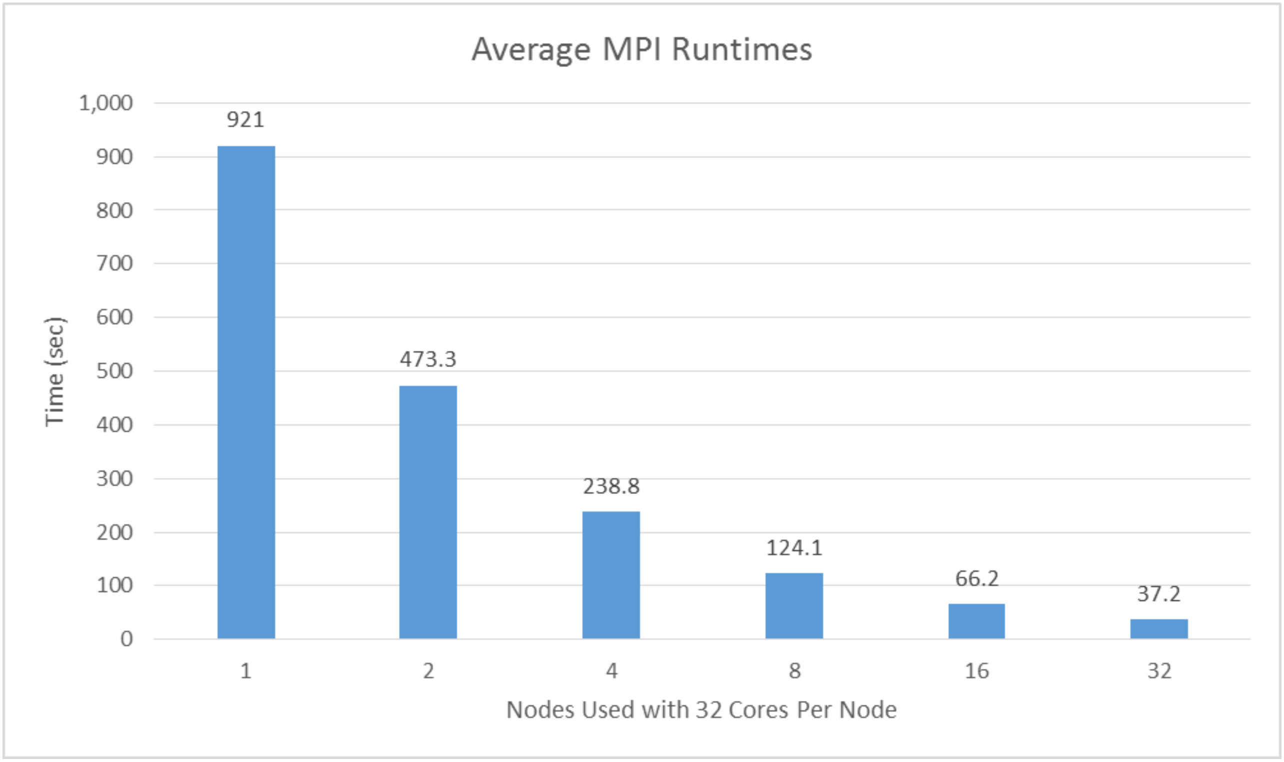

Figure 2 - Average Runtimes vs. Nodes Used with MPI

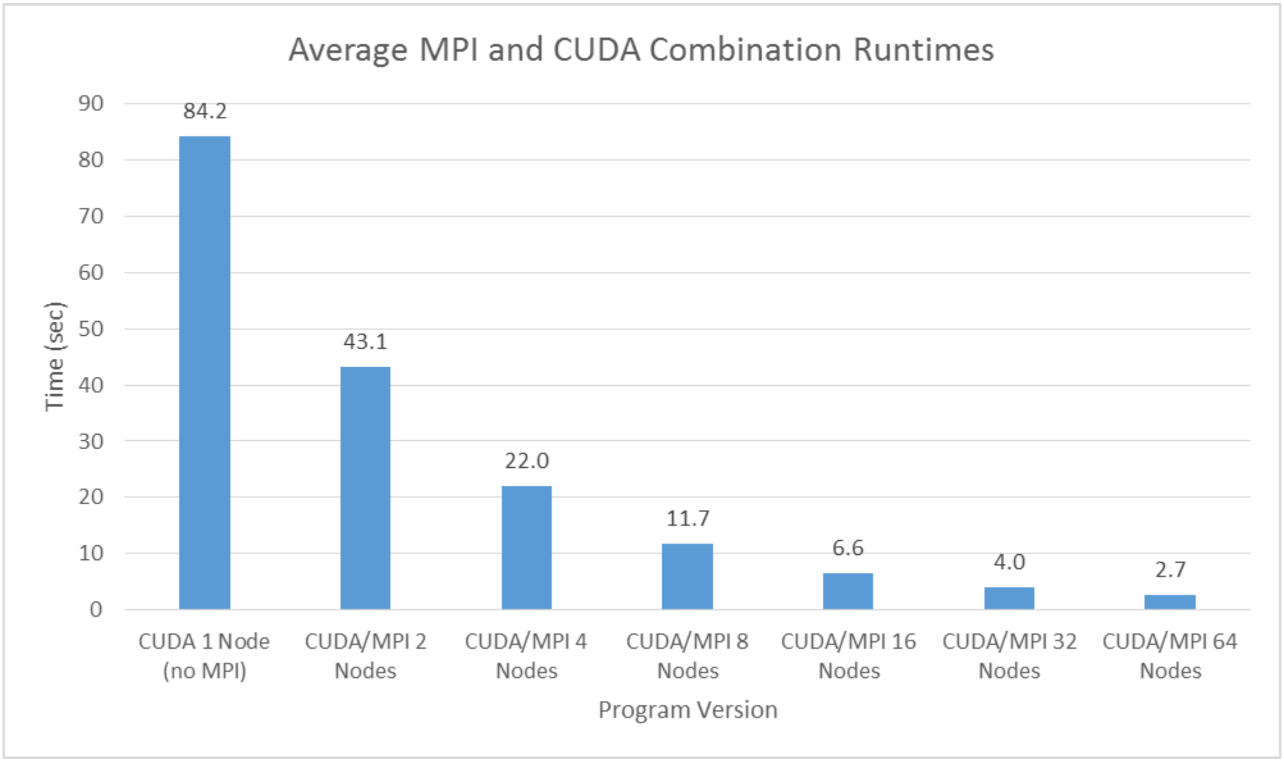

Figure 3 - Average Runtimes vs. Nodes Used with MPI/CUDA Hybrid

6. CONCLUSIONS

During the internship, we developed additional versions of a program to test graphs for the party problem. We were able to develop instructions to compile and the run the programs on Blue Waters and conduct performance testing. These things all have made the existing curriculum module more useful.

7. REFLECTIONS

This experience has been special to me in many ways. I think it is a wonderful opportunity to allow undergraduates a chance to work one-on-one with a mentor doing some research project, especially this project which allowed me the unique opportunity to work on the Blue Waters supercomputer. I loved being able to not only work on Blue Waters but also getting to see it in person and learn how to use it. The Petascale Institute was amazing, meeting other students and having the ability to dedicate two weeks to learning about the system and parallel computing. Learning about the different libraries was extremely useful to me since I wrote and used code in almost every library and hybrid we discussed.

The Petascale Institute allowed me, as a computer science major, to expand my knowledge base beyond what I had learned in the classroom. I am now more confident with using remote systems, Linux command and shell scripts. Even without any previous knowledge about parallel languages or programming I left the institute with a general understanding and the internship itself has allowed me to help others learning parallel too. The internship this year encouraged me to take a class in parallel at my school and I felt that I gained even more knowledge from that class and was also able to help those who struggled because of my experience with Blue Waters.

In the field of computer science there are a lot of options for career paths. I came into this internship no knowing what really interested me specifically in computer science. But the work I did this year made me realize why I love computer science so much, I love solving puzzles and figuring out how the pieces work together. All the Party Problem code in different libraries each required a different understanding of parallelism, and getting them to work together was an even bigger challenge, but with lots of guidance from my mentor we figured them out and got some interesting results. I also discovered how interesting parallel computing is to me. I still have another year of college left, so I am not ready to decide where I go from here; but, I know now I would enjoy working on parallel in the future. I feel like it is a growing field and now I have a unique experience thanks to this internship.

8. ACKNOWLEDGMENTS

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. We thank the Blue Waters Student Internship Program for providing Ruth with this opportunity. Finally, we thank the University of Mary Washington, which provided Ruth with room and board for the summer through their Summer Science Institute and funding for the wet-lab studies.

9. REFERENCES

[1] ramsey theory - Wolfram|Alpha. (2012). http://www.wolframalpha.com/input/?i=ramsey+theory.

[2] Weisstein, Eric W. "Ramsey Number." From *MathWorld*--A Wolfram Web Resource. http://mathworld.wolfram.com/RamseyNumber.html.

[3] S. P. Radziszowski, Small Ramsey Numbers, The Electronic Journal of Combinatorics. DS1.10. (originally published July 3, 1994, last updated August 4, 2009), http://www.combinatorics.org/ojs/index.php/eljc/article/view/DS1/pdf.

[4] D. Toth and M. Bryant, A Performance Comparison of a Naïve Algorithm to Solve the Party Problem using GPUs, Journal of Computational Science Education, v. 3, issue 2, December 2012.

[5] http://www.shodor.org/petascale/materials/UPModules/howManyPeople/

[6] http://www.shodor.org/petascale/materials/modules/

[7] S. Walkowiak, K. Wawruch, M. Nowotka, L. Ligowski, W. Rudnicki, Exploring utilization of GPU for database applications, Procedia Computer Science 1(2010) 505-513.

[8] V. W. Lee, C. Kim, J. Chhugani, M. Desiher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, P. Dubey, Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU, Proceedings of the 37th annual international symposium on Computer architecture (2010) 451-454.